Coupon Recommendation Prediction Task: COMPSCI 671 Kaggle Competition Report

Karthikeyan K
Kaggle username: kkarthi

December 10, 2021

Abstract

This is a tech report describing several approaches used for the kaggle competition with Coupon Recommendation dataset (**prediction track**). The model that worked best is ensemble of three class of models (1) Neural networks, (2) Random Forest and (3) Extreme gradient boosting. Within each class of models, I trained several instances with different hyper-parameters and averaged their prediction probabilities using a exponential weighing scheme. The final best test accuracy I got is **78.933%** (based on kaggle score) and best validation accuracy I got is **80.8%**.

1 Introduction:

In Coupon Recommendation dataset, the task is to predict weather a user accepts a coupon before it expires using his/her demographic and preference data as well as the context and coupon information. Intuitively, the task is very challenging because (1) under similar circumstances, different people might behave differently and just demographic or preference data may not be enough (but could be useful) to characterize each individual. (2) there is inherent (probably high) randomness as even under similar circumstances, the same user might behave differently (just based his/her mood). It is an interesting dataset as we can understand the influence of user demographics and circumstances on ones behaviour.

I approached the above decision problem using several approaches from simple models like K-nearest neighbours, Decision Trees, Logistic regression to more complicated models like neural networks, random forests and extreme boosting which are described in detain in § 3. In general, I found that more complex models performed better. [1]

2 Exploratory Data Analysis

Visualize each feature: From figure 1, we can easily visualize the unique values (categories) in each features along with their counts. A simple observation is that most features have only a few category except some features like occupation, income, age.

[1]Github Link for code: https://github.com/keyank/Karthikeyan-671-Kaggle-Competition

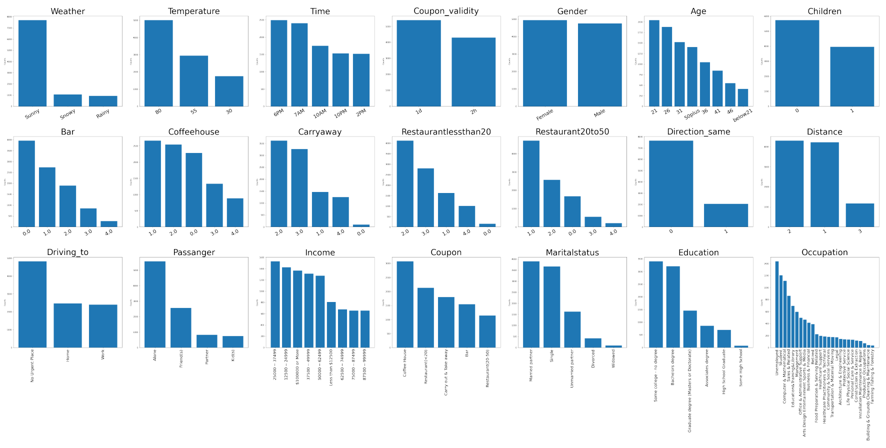


Figure 1: Distribution of each feature

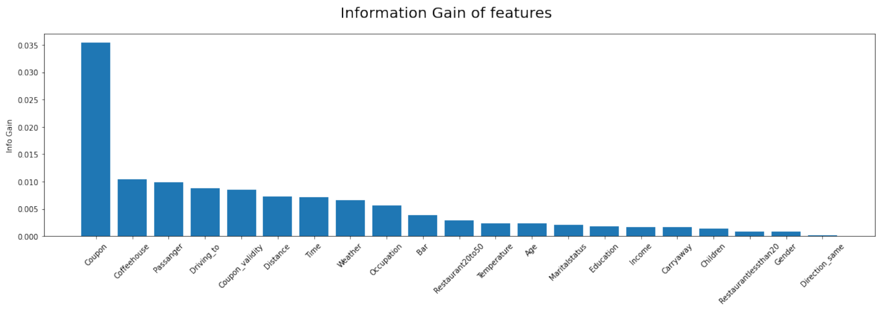


Figure 2: Information Gain of all the features

Information Gain If I can use just one feature to predict the output, then what feature should I choose? The one that gives highest information gain [Kent, 1983]. Although information gain, does not tell us the complete picture they still are fairly informative (that's the reason we use info-gain to decide the feature to split on in Decision trees). Figure 2, shows the information gain of all the features in decreasing order, we can see that some features are highly informative (the information gain of binarized features are in the appendix). However, the caveat is that two features can have a very less information gain, yet when combined (i.e. given both the features) their information gain could be very high.

Variable Importance: Model Reliance (MR) and Conditional Model Reliance (CMR) are useful only to understand a particular trained model. They don't tell us anything about the how useful a feature is for a given algorithm or for a class of models. Algorithmic Reliance and Model Class Reliance are two variable importance technique that are quite useful in this context.

Algorithmic Reliance (AR) tells us how important is the feature for the given algorithm with the given dataset (The results will could be different for different algorithms). Again, there is a caveat in AR as well, assume that we have 2 features that are highly dependent, therefore either one of them is sufficient. When only one of them is removed, we may not see a huge decrease in accuracy, but removing both of them could highly decrease the accuracy.

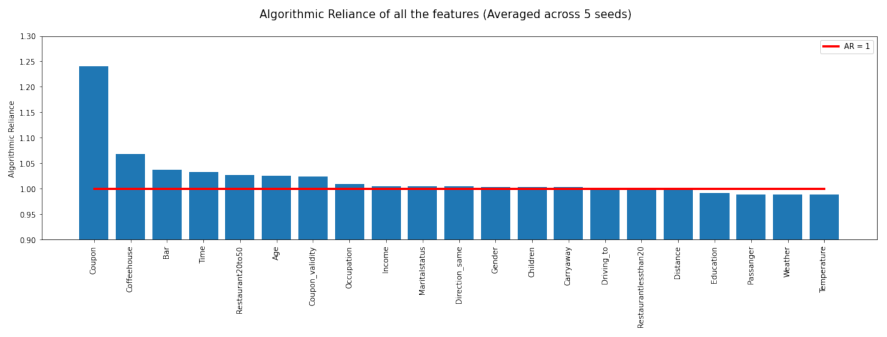


Figure 3: **Algorithmic Reliance For Random Forest** : The results are averaged across 5 random seed with a fixed hyper-parameters

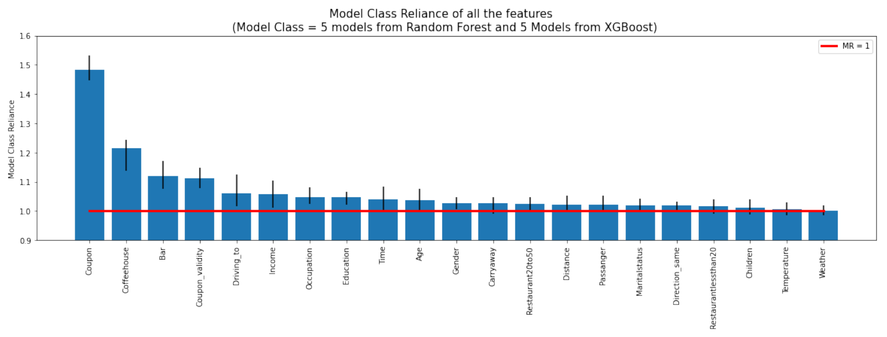


Figure 4: **Model Class Reliance for Random Forest and XGBoost:** Bar show the average Model reliance across the models in model class (5 best models from each of Random Forest and XGBoost). Error bars show the minimum and maximum MR

To be clear, in this context, by algorithm, I mean an algorithm (like Random Forest, Decision Tree) with a fixed hyper-parameter (but can have different random seeds). The AR for Random Forest with a fixed hyper-parameter (the one that gave best test accuracy) is shown in the figure 3. Note that I ran the algorithm with 5 seeds and averaged the errors (or accuracy) and used the average accuracy before and after removing the feature to calculate the final AR.

Model Class Reliance (MCR) MCR [Fisher et al., 2019, Fisher et al., 2018] helps us understand how any good predictive model (from a model class) depends on the feature (minimum and maximum Model Reliance). To measure MCR, first we need to define a model class and a reference loss (or error) with tolerance rate (ϵ). I used the random forest and XGBoost models with best 5 hyper-parameters as my class of models. I used a reference accuracy of 0.75 validation accuracy (error is 0.25), and all the 10 models get better validation accuracy than that, therefore all of them would be used to calculate CMR (even $\epsilon = 0$ works for here). Figure 4 shows the model class reliance. The bars show the average MR of 10 models and the error bars show the minimum and maximum MR among the 10 models in the chosen model class. I did not include neural networks in my model class, because its not fair to compare the validation accuracy of Neural networks with that of Random Forest or XGBoost Model (refer § 4.1).

Feature Description: I understood the features by reading the description from the paper [Wang et al., 2017]. I am skipping the description here, please refer *Appendix B. Mobile Advertisement data sets* of [Wang et al., 2017]. I will briefly describe a couple of features that are very intuitively and help us predict the label.

- Driving to No Urgent Place: +1; to Work or home : -1
- Weather: Rainy, Snowy = -1; Sunny = +1 (similar with Temperature)
- Time: late night or early morning = -1, afternoon = +1

Why do we care about AR, MCR or Features? When we are working only on the predictive task, the role of analysing AR, MCR or intuition behind how features relate to labels are may not seem very useful. However, for interpretability these quantities would be very useful. We can possibly select only a few features using Information Gain, AR or MCR to train our model (fewer the features, better the interpretability). We can possibly come up with simple rule based approaches (or verify the rules learn from some rule mining algorithm) from the intuitions of how a feature would relate to label. These intuitions could possibly help us even understand neural models (along with the explanations from methods like LIME). Note that in this work, my objective is to (only) come up with better predictive models and not about interpretability.

Demographic Features or Contextual Features? Its very interesting to understand weather demographic (and preference features) or situation (or context) play a huge role in determining a user's behaviour. To answer this question, I trained two random forest models one using only the demographic features and other using only the contextual features. The model with only demographic feature gives 61% accuracy, while the mode with only situation feature gives 67.3% accuracy, indicating that situation or context is more important.

Initial Thoughts on what might work:

1. All the features are categorical, therefore, tree based often models work.
2. I converted the features to binary one hot vectors because the binary features are better to work with especially for decision boundary based models like logistic regression or neural networks. I did not normalize because the features are already binary.
3. Neural networks might tend to heavily over-fit, because the number of data points is relatively less. Moreover, the data has inherent noise (refer §1), so we should try to avoid over fitting. High regularization (dropouts) might be required.
4. Not all the features are equally important, therefore K-Nearest neighbour(KNN) models or SVMs with RBF Kernels may not work. However, we can try to weight the features according to their importance and then use KNN or SVN with RBF Kernels (equivalently, SVMs with Automatic Relevance Detection Kernels).
5. Ensembling the results might work better. I tried with different models and the predictions are quite different. When we have a diverse set of predictions then ensembling could provide a more robust estimates.
6. Instead of removing the rows containing 'nan', we can replace the 'nan' with one of the category (say 0) from that feature, so that we get more data for training as well as we can handle 'nan' better in testing.

Model	5 Fold Validation Accuracy
Logistic Regression	68.7
Decision tree	70.3
RandomForest	**76.0**
Adaboost	73.9
K-Nearest Neighbors	72.3
Naive Bayes (BernoulliNB, MultinomialNB and CategoricalNB)	65.9
Linear and Quadratic Discriminant Analysis	71.2
Bagging Classifier	75.5
Gradient Boosting (sklearn)	74.2
XGBoost	**76.3**
**Fully connected Random Neural Network (with PyTorch)	**80.8**
**Fully connected Neural Network (with PyTorch)	**80.0**
*** Denoising Auto-encoder Neural Network (with PyTorch)	-

Table 1: **Model Exploration with simple hyperparameter Fine tuning:** For each of them I tuned the hyper-parameter to some extent but not highly. ** NN results are with 1 of the 10 fold CV (not averaged) and best among across all the epochs. *** The results with denoising autoencoders are similar to simple NN

3 Methods

Initial Model Exploration using sklearn (and PyTorch): It is generally advisable to first explore different models (from simple models to complicated models) before exploiting any particular model and sklearn library makes it really simple to explore many different algorithms. I tried with all the models in Table 1 using sklearn (and PyTorch). From basic exploration, I found that RandomForest, XGBoost are promising. I got comparable results with Neural Network as well but not as good as the 80%, to get that, I had to tune a lot. I explain each of the 3 models in detail in next part of this section.

3.1 Random Forest

Random forest (RF) [Liaw et al., 2002] uses the idea of Bagging, which is, if we average the predictions from multiple well trained models, it reduces the variance in prediction. RF extends simple bagging by selecting only a subset features for each split in every decision trees.

Why did I choose: From Table 1, we can see that Random Forest is quite promising. Moreover, since the features are categorical (converted to binary) tree based models might work better. Random Forest uses prediction from multiple Decision Trees and often outperforms simple Decision Tree. However, on the downside, this makes the model less interpretable. [2]

[2]Although sklearn's Bagging Classifier is also quite promising, is quite similar to Random Forest, therefore I did not consider as another methods.

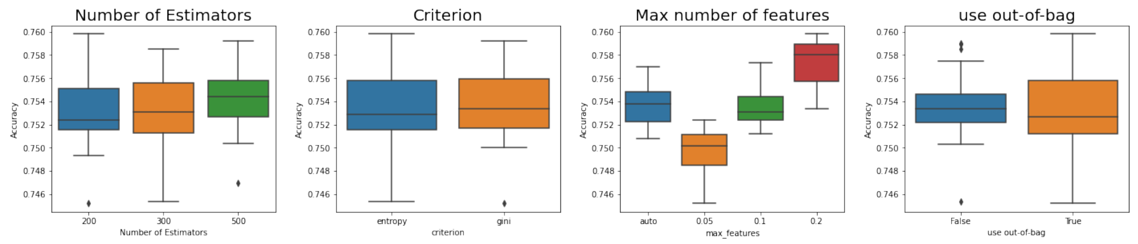


Figure 5: **Random Forest Hyperparameters vs 5 fold CV Accuracy:** The figure shows the distribution of accuracy for each hyperparameter value. While analysing one hyperparameter, I let other hyperparameter changes freely. i.e. when calculating accuracies for number of estimator (= 200), criterion, max features, oob score takes all possible values.

Training: I trained the model using sklearn's library, which makes it really simple and efficient to train and test models. To get better model (1) I wanted to choose good hyper-parameters (need validation set) and (2) use all the data for training (should NOT keep aside any data for validation). I can accomplish both by with the following two step training process, For a given hyper-parameter, in step (1), I split the entire data into multiple train and validation sets using 5 fold cross validation and then use the average of 5 validation accuracies to measure how good the given hyper-parameter is. In step (2), I retrain the model using the entire data with the given hyperparameter, and use the new model to make predictions on the test data and store both the predictions (probabilities) and the average CV accuracy to use later for final ensembling. I weigh the predictions based on the average CV score for final ensembling (refer § 3.4)

Time Consumed: Training time heavily depends on the hyper-parameter (in particular number of estimators). With one hyperparameter, for both step 1 and step 2 combined, it takes about ≈ 18 secs with 500 estimators, while it only takes ≈ 4.6 secs with 100 estimators (≈ 7.77 and ≈ 11.34 with 200 and 300 estimators respectively). For my final ensemble, I used about 50 models with different hyperparameters for which it takes about 600 secs.

Hyper-parameter Tuning: My hyper-parameter tuning also strategy also follows a two stage process. The first stage is manual exploration. I start with the default parameters and tune only one hyperparameter at a time, once I find a reasonably good value for that I tune the second hyperparameter and continue. After the initial exploration, I create a range for each of the hyper-parameter that I wanted to explore. For each of the hyperparameter, I train the model, and store the predictions and average CV accuracy for that hyperparameter.

I tuned (1) number of estimators (2) criterion (3) maximum features (4) oob score (weather or not to use out of bag samples to estimate generalization scores). I also tried with different class weight to balance the positive and negative class but it seemed to be bad. My final model consist of 48 different random forest models with n_estimators $\in \{200, 300, 500\}$, criterion $\in \{entropy, gini\}$, max_features $\in \{auto, 0.05, 0.1, 0.2\}$ and oob_score $\in \{True, False\}$. Refer Figure 5 for hyperparameter vs accuracy distribution. Maximums are the most interesting.

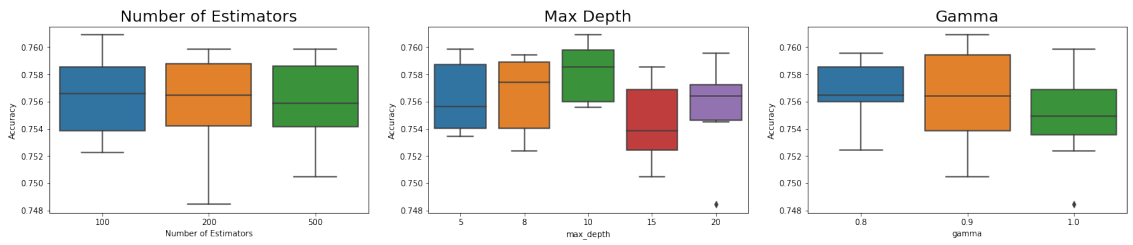


Figure 6: **XGBoost Hyperparameters vs 5 fold CV Accuracy:** Similar to figure 5

3.2 XGBoost:

In general, when nothing works boosting works, boosting is such a powerful algorithm, we can do boosting with Decision trees. I first tried with Adaboost using sklearn, I was able to increase the performance of Adaboost by using a a slightly more deep decision trees than the default (depth=1). However, I observed that XGBoost [Chen and Guestrin, 2016] is performing much better than Adaboost; its even better than Random Forest.

Why did I choose: From Initial exploration, I found that XGBoost is performing better. Moreover, XGBoost[3] library is also simple to use (just like sklearn). Again XGBoost also uses trees (which are good for categorical data).

Training: Training procedure follows exactly the same two step process described in Random Forest, except that I use XGBoost library instead of sklearn.

Time Consumed: Time depends mainly on number of estimators and depth. It takes about $\approx 9.4, 18$ and 43 secs for $100, 200$ and 500 estimators (with depth 10) respectively. It consumes about $\approx 4, 7, 9.4, 16$ and 20.5 secs for depth $5, 8, 10, 15$ and 20 respectively. I used a total of about 39 models. Together it takes about ≈ 1250 secs

Hyper-parameter Tuning: Tuning strategy is same as described in RF. For XGBoost, I tune (1) Number of estimator (2) maximum depth of trees and (3) Gamma (Minimum loss reduction required to make a further partition on a leaf node of the tree). My final model consist of 39 models with $n\_estimators \in \{100, 200, 500\}$, $max\_depth \in \{5, 8, 10, 15, 20\}$ and $gamma \in \{0.8, 0.9, 1\}$ (when number of estimator is 100, I don't use depth of 5 and 8). Refer Figure 6 for distribution of accuracies with various hyperparameters.

3.3 Neural Networks (NN):

Neural Networks are known to be one of the most powerful machine learning algorithms. It is known that a two layer NN can approximate any function. However, NN are notorious for over-fitting, especially with limited data, therefore we need to use definitely use some regularisation. I use high dropouts.

[3]https://xgboost.readthedocs.io/en/stable/

Why did I choose: Again, From Table 1, we can see that NN results are very promising. Instead of using sklearn's MLP, I implemented my own NN using PyTorch, because I can use GPUs with PyTorch (I modified the code to experiment with denoising autoencoders).

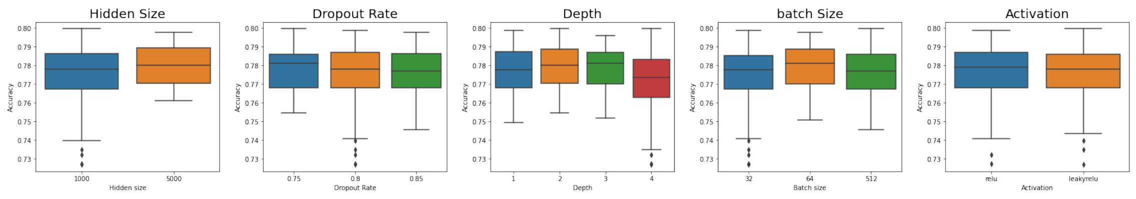
Training: The two step process used for RF and XGBoost does not work for NN, because we need to choose the best model among all epochs (it is a relatively bad to fix the number of epochs and use the final model). I use the following procedure to get maximum benefits, i.e. use all the data (for training) as well as chose the best hyperparameter. First I split the data into 10 train and validation folds. For each fold, I train the model for 100 epochs using the given hyperparameter. After each epoch, I test on validation set and keep track of the best model (the one with beat validation accuracy). Finally I use the best model(s) to predict (probabilities [4]) on the test set. Note that for each hyperparameter setting, I will have 10 different NN models (trained using 10 different subsets of data). Therefore, we have used all the data for training and we were able to select the best model among different epochs as well as get a score for each model (score = best validation accuracy) that we can use during the final ensemble.

Time Consumed: Time consumed depends heavily on the batch size, number of layers and the hidden size. To give a brief idea, for single model (1 of the 10 folds) it takes about 11 and 12 secs for 1000 and 5000 hidden units (with for single hidden layer and 512 batch size). In comparison, takes about 50 secs with 3 hidden layers (same 512 batch size). Whereas with a batch size of 32, it takes about 5 mins (with 5000 hidden units and 3 layers). I trained a total of 500 models and 460 models with random neural networks and simple neural networks (described below), together it consumes about 3-4 GPU hrs for each (both done in parallel with 2 different GPUs)

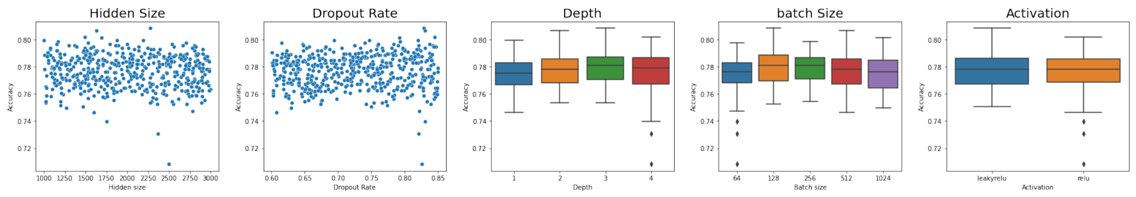
Hyper-parameter Tuning: First I do a manual exploration to find the set of hyperparameters to explore and their range. Once I find a reasonable range, I follow two different approaches (1) Random Networks and (2 Simple NN. I tune the following parameters using both the approaches (1) number of Hidden units per layer (same for all hidden layers) (2) Dropout rate (3) Depth of the network (4) Batch Size, (5) Activation function.

- Both are fully connected NN. The only difference is how we choose hyperparameters.
- **Random Networks:** To train each model, I randomly choose a set of hyperparameter (independently) and the train the model with that. Even within in 10 folds, each model can have different architecture. **Pro:** More exploration, therefore, atleast some models could be really good (better maximum accuracy) **Cons:** Some models could be bad (bad minimum accuracy), but we can just throw away bad models.
- **Simple NN:** I fix a set of hyperparameters beforehand and train the model with each of them. For each hyperparameters, we train 10 models, one corresponding to each fold. **Pro:** Most models are good. **Con:** Less exploration, therefore maximum accuracy may not be as good as random networks but minimum accuracy would be much better.

[4] I use the softmax output on final layer as a surrogate for probabilities



(a) Simple Neural Network



(b) Random Neural Network

Figure 7: **Neural Network Hyperparameters vs Validation Accuracy**

Hyper parameter tuning Observations

- Higher the dropout rate better the generalization [Srivastava et al., 2014]. However, high dropouts also mean under-fitting, therefore, we need to train for more epochs (slower but better convergence).
- Since the data is sparse, we need much higher dropout rate than usual. (There is only about 18% 1s, the rest 82% all 0s). As a rule of thumb, we can start around dropout rate of 0.82.
- Higher dropout rate also mean we need more number of hidden units (to avoid under fitting).
- NN with depth greater than 5 seems a bit bad.

3.4 Final Ensemble

Step 1: This is done for each model class (Random Forest, XGBoost and NN) separately. Let there be m models and let $s_1, s_2, ...s_m$ be their scores (average validation accuracy). Calculate the normalized weights of each model as follows $w_i \propto exp(\lambda(s_i - \hat{s}))$, where $\hat{s}$ is the mean score and λ is a parameter to choose. I choose lambda such that it roughly follows Pareto Principle – top 20% of model gets 80% of weight. For a given input x, let $p_1, p_2, ...p_m$ be the probabilities of 1, predicted by each model respectively. The final aggregated prediction would be $\sum_{i=1}^{m} w_i p_i$ (this can be thought as predictive posterior probability).

Step 2: For a given input x, let p^{rf}, p^{xgb}, p^{snn} and p^{rnn} be the probability of 1 predicted by the aggregated random forest, XGBoost, Simple and Random NN respectively. Find the average probability $p = \frac{1}{4}(p^{rf} + p^{xgb} + p^{snn} + p^{rnn})$. If $p \geq \delta$ predict 1 otherwise predict 0. Usually, $\delta = 0.5$, but $\delta = 0.49, 0.485$ gave slightly better test results.

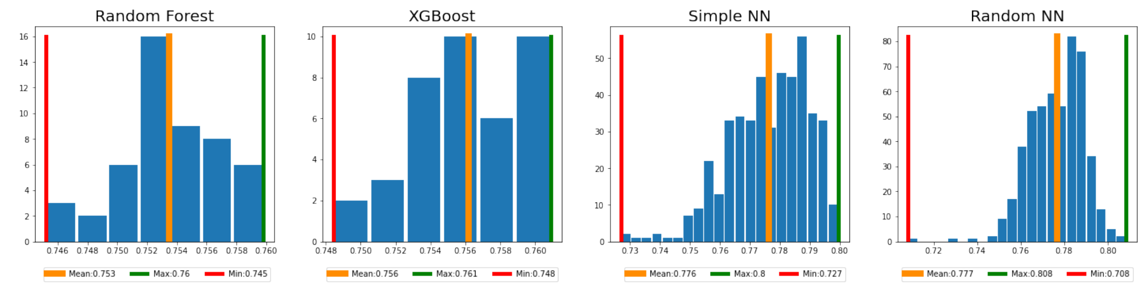


Figure 8: **Distribution of Accuracy from Various Model classes**

4 Results

4.1 Prediction

Figure 8 shows the distribution of validation accuracies by each of the model class (Min, max and mean are also shown in the same figure at bottom). We can observe that Random NN and simple NN gives the overall best validation accuracy of 80.8 and 80.0 respectively. However, this is not directly comparable to Random Forest or XGBoost because (1) These are validation accuracy of each fold (not average) (2) Even the average CV accuracy won't be comparable because, we are choosing the model with best validation accuracy among all the epochs (so, we can expect that the validation accuracies from NN would be slightly higher).

4.2 Fixing Mistakes

What I tried but did not work: I tried denoising autoencoder where the neural network is first pretrained to reconstruct the corrupted input and then the model is finetuned to predict the label. To understand the motivation, consider a linear classifier, if a data point is on far from the classification boundary, then the classifier might predict a label with high probability score (say 0.99). However, if this data point belongs to a low data density region, then the model's prediction can not be trusted. Motivated by the above, I thought if the model is not able to reconstruct the input properly, then probably I won't trust its prediction on the input (and use some other models for those inputs). However, this hypothesis does not seem to hold as good. There is some correlation between high loss in reconstruction of input and the predicted label to be wrong, however, the correlation is small. I think it would be interesting to investigate further in this direction. I thought about using the discriminator part of Generative Adversarial Networks (if it predicts the input as fake then I won't trust its prediction), but I left it for future work due to time constraints.

I tried with simple K-nearest Neighbours, but it did not work, which I expected because all features are not equally important (We know from AR, MCR or Information gain). I hoped that weighing the features according to their importance (information gain) would increase their performance, however, I did not see any improvements, I thought to try SVM with RBF kernels after weighting (ARD kernels) but since it did not work with KNN I think it doesn't work with SVMs as well.

Deeper NN performed worse, I thought to try deeper NN with residual connections. I would explore that in future works.

References

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

[Fisher et al., 2018] Fisher, A., Rudin, C., and Dominici, F. (2018). Model class reliance: Variable importance measures for any machine learning model class, from the "rashomon" perspective. *arXiv preprint arXiv:1801.01489*, 68.

[Fisher et al., 2019] Fisher, A., Rudin, C., and Dominici, F. (2019). All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *J. Mach. Learn. Res.*, 20(177):1–81.

[Kent, 1983] Kent, J. T. (1983). Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173.

[Liaw et al., 2002] Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

[Wang et al., 2017] Wang, T., Rudin, C., Doshi-Velez, F., Liu, Y., Klampfl, E., and MacNeille, P. (2017). A bayesian framework for learning rule sets for interpretable classification. *Journal of Machine Learning Research*, 18(70):1–37.

A Appendix

Information Gain with Binary Features

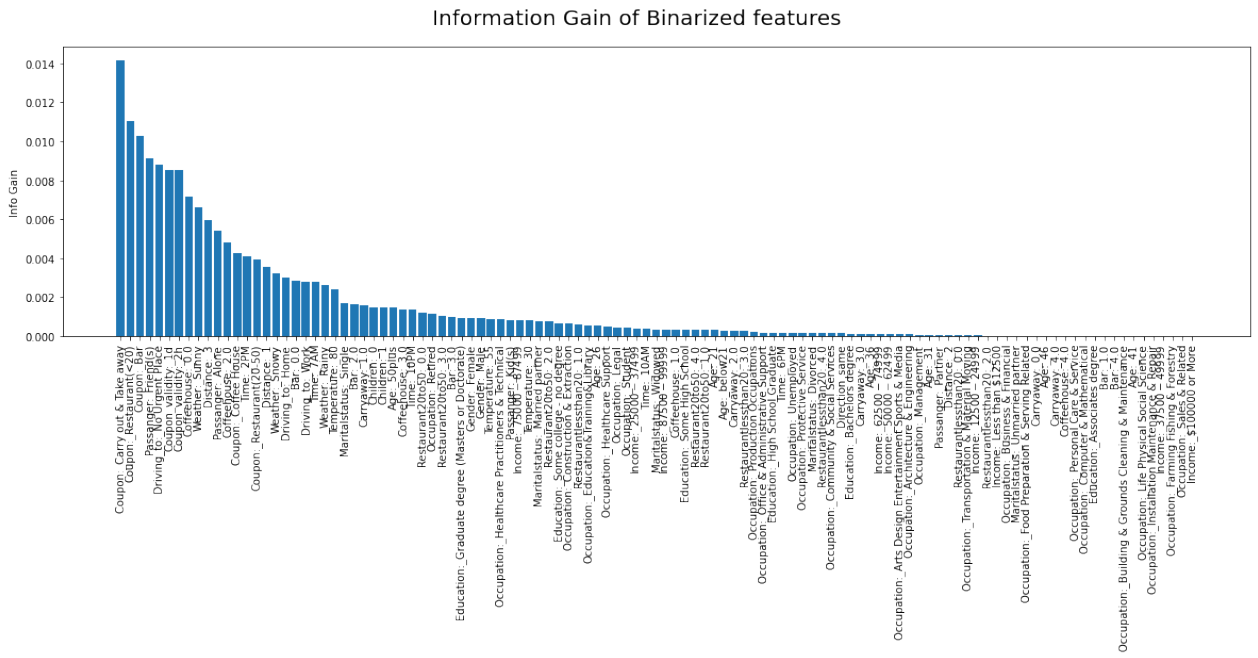


Figure 9: Information Gain of all the binary features

Challenges with Visualization the entire data (PCA) I tried to visualize the features (X) by projecting it to 2D plane using PCA (and then plotting the data from different classes with different colors). However, the results does not seems much useful. I also tried to project the joint data (X, Y) into 2D plane and visualize again, again the results are not very useful.

Code

```
Github Link: https://github.com/keyank/Karthikeyan-671-Kaggle-Competition

I have also attached the code below.
```